# **Aspect-Oriented Programming in PHP**

**FLOW Framework**

FLOW is an enterprise application framework. It can be used as a standalone full-stack framework for applications. It introduces some new concepts of software development that have not been adapted to PHP before, such as aspect oriented programming. FLOW's AOP framework allows you to use the popular AOP techniques in your PHP application.

**What is aspect oriented programming(AOP)?**

In terms of software development, it means that the classes should only care about one thing and the business logic should be separated from additional services you may want to implement in your application. By separating the concerns you are making the software easier to maintain by grouping features and behavior into manageable parts which all have a specific purpose and business to take care of.

### **FLOW AOP concepts**

FLOW implements a reduced set of AOP, which satisfies most needs of web programming. FLOW makes use of PHP's reflection capabilities to analyze declarations of aspects, pointcuts and advices and implements method interceptors as a dynamic proxy. The proxy classes are generated automatically by the AOP framework and cached for further use. If a class has been adviced by some aspect, the Object Manager will only deliver instances of the proxy class instead of the original. This approach provides the advantage of dynamic weaving with a minimum performance hit.

**Aspect**

An aspect is the part of the application that cross-cuts the core concerns of multiple objects. In FLOW, it is implemented as a regular class tagged by the *@aspect* annotation.



**Advice**

An advice is the action taken by an aspect at a particular join point, the implementation of the concern you want to weave into a target. It is implemented as a method of the aspect class. The method can be executed before and/or after the join point is reached.

There are four types of advices for four different kinds of interception:

* **Before**: executes code before the target method is invoked. It is declared by the *@Flow\Before* annotation
* **After returning**: executes code after the target method normally (without throwing an exception) returns from execution. It is declared by the *@Flow\AfterReturning* annotation.
* **After throwing:** executes after the method returns with an exception. It is declared by the*@Flow\AfterThrowing* annotation.
* **After:** executes after the target method returns, either normally or with exception. It is declared by *@Flow\After* annotation.
* **Around:** may decide to call the original method or not and even modify the result of the target method or return a completely different one. It is declared by the *@Flow\Around* annotation.

**Join point**

A join point is a point occurring in the flow of the program, such as the execution of a method or the throw of an exception, not a definition which defines that point. In FLOW, it is represented by the *Neos\Flow\AOP\JoinPoint* object containing information about the circumstances (name of the called method, passed arguments, type of the exception).

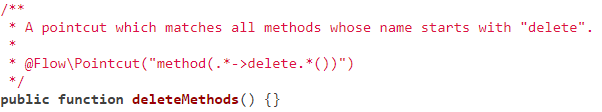
**Pointcut**

A pointcut defines a set of join points which need to be matched before running an advice.

You can define a pointcut in an advice declaration or set up a named pointcut.

A named pointcut contains two pieces of information: the pointcut name, defined by the method name, and the pointcut expression which defines the condition under which an advice should be executed, declared by an annotation.

Pointcut expressions can be combined into a single one by using the *&&*(and), *||*(or) and *!*(not) operators.



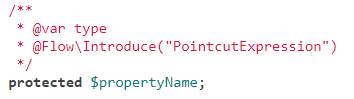
**Target**

A target class or method is a class or method being adviced by aspects.

**Introduction**

An introduction redeclares the target class to implement an additional interface. By declaring an introduction, you may introduce new interfaces and implementations of required methods without modifying the original class's code. Also, introductions can be used to add new properties to a target class. There can be different kinds of introductions, such as

* interface introduction: 
* trait introduction: 
* property introduction:



**How can AOP help you**

Imagine you want to log a message inside methods of a class. Doing this in a lot of places, logging will become a part of the class’s logic. With AOP, the class would know nothing about logging, concentrating only on the business logic.

So, by using aspect oriented programming, the source code will be easier to read and understand, repetition will be decreased, resulting in shorter development time and fewer defects. Not only less-experienced developers will appreciate AOP, not having to learn how to weave these cross-cutting concerns throughout their code, but also the experienced developers will be able to focus primarily on implementing the proper business logic, due to the more fluid coding.

Neagos Bianca

Group 931